ULTRA LONG-LIFE AVIONICS ARCHITECTURE

Savio Chau, Abhijit Sengupta, Tuan Tran, Alireza Bakhshi and Tooraj Kia
Jet Propulsion Laboratory
California Institute of Technology
Pasadena, California, USA

ABSTRACT

For survival and achieving reliability in ultra long-life missions, fault tolerant design techniques need to handle the predominant failure mode, which is the wear-out of components. Conventional design methodologies will need excessive redundancy to achieve the required reliability. The objective of this paper is to present a new approach to design a more efficient fault-tolerant avionics system architecture that requires significantly fewer redundant components. This architecture uses generic function blocks that can be programmed, in-flight, to replace a wide variety of components. Effectively, each individual generic block is almost equivalent to an entire redundant string of components used in conventional approach. As a result, the ultra long-life system can achieve much higher level of reliability while carrying far fewer components. Due to the programmable nature of generic redundant blocks, the physical location of a specific component is not pre-determined and accordingly, wireless interconnection is employed to provide the necessary flexibility in connectivity. A testbed of this architecture is being developed at the Jet Propulsion Laboratory.

INTRODUCTION

After extensive exploration of Solar System, NASA is close to completion of the initial reconnaissance, and has started landing and sample return missions on many planets, satellites, comets, and asteroids. The next logical step for space exploration is to expand the frontier to the far reaches and beyond the solar system with possible missions to include Pluto and Kuiper Belt objects sample return or interstellar space exploration. Such missions will typically last for about 30 to 50 years. The current technologies and spacecraft design techniques are inadequate to support such long life missions. Conventionally, spacecraft avionics are designed with fault-tolerant capabilities to tolerate random failures, which are typically handled by dual or triple redundancy for a relatively short mission life. In comparison, the predominant failure mode in an ultra long-life system is the wear-out of components - active components in the system are destined to fail before the end of the mission. Therefore, using current technologies, an ultra long-life system will necessarily require a large number of redundant components. This would be impractical under conventional fault tolerant systems as the current techniques are not very efficient when applied to more than double redundancies.

In addition to deep space exploration, many terrestrial spacecrafts also require ultra long life design. One of the major factors in the maintenance cost of communication satellite networks is the replacement of failed spacecrafts. The total cost for manufacturing, testing, and launch to replace a failed satellite exceeds hundreds of millions of dollars. In addition, there are also costs associated with the lost of services during a down time.

Ultra long-life space systems need breakthrough technologies in four main areas [1]:

- long-term survivability – to handle failures due to random events, design errors, and wear-out mechanisms;
- optimal management (administration) of consumable resources – to maximize the acquisition and to minimize the consumption of consumable resources such as power, fuel, and the generic blocks;
- evolvability and adaptability – to have built-in mechanisms so that the capabilities and functions of the spacecraft can be upgraded, as and when needed, after launch; otherwise, the useful life of the spacecraft will be limited by the obsolescence of the on-board technology;

- long-term operation of the spacecraft – to reduce the operation costs and maintain a workforce knowledgeable of the spacecraft.

The following discussion will focus on the architectural aspect of the long-term survivability.

AN AVIONICS ARCHITECTURE FOR ULTRA LONG LIFE MISSIONS

It is simple to note that current technologies and spacecraft design techniques are inadequate to support a long-life mission. In a dual-string architecture, the system fails when the processors in both strings have failed. It is impossible to use the other components in the system, which might still be functioning correctly, to revive the system. Consequently, the improvement of system reliability by the conventional fault tolerance architecture will diminish in time. This can be illustrated by reliability modeling as shown in Figure 1.

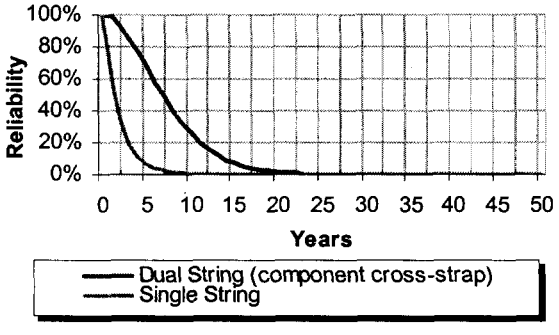

Figure 1: Reliability of Single and Dual String Systems

Since the predominant failure mode in an ultra long-life system is the wear-out of components, without an efficient utilization of redundant components, too many such components are needed to achieve high reliability.

The NASA Exploration Team has developed a highly reconfigurable avionics system architecture that uses redundant resources more efficiently. This architecture employs *Generic Function Blocks* as illustrated in Figure 2. Such a generic function block can be configured or programmed in-flight and can replace a wide variety of functional blocks, once such blocks have been diagnosed to be faulty. Hence, in some sense, each individual generic block is almost equivalent to an entire redundant string of components in the conventional approach. Accordingly, the ultra long-life system can achieve much higher level of reliability while carrying much less redundant components.

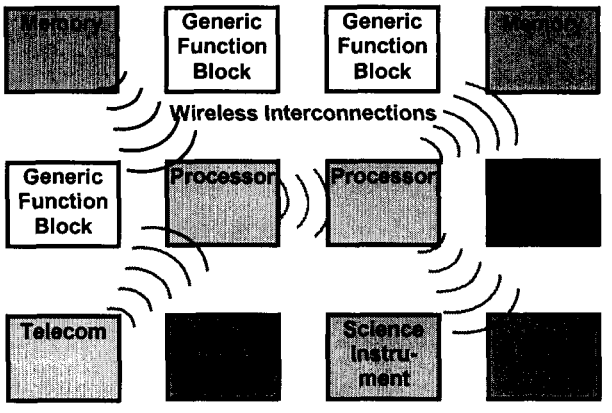

Figure 2: Avionics System with Generic Function Blocks

A reliability model shown in Figure 3 illustrates the reliability of the fault-tolerant architecture using Generic Function Blocks. It compares a dual-string system, in which each string has 8 components, to an ultra long life avionics architecture consisted of 16 *Generic Function Blocks*, assuming the Mean-Time-Between-Failure of each component is 125,000 hours (14 years). It is clear that the *Generic Function Block* architecture has much higher reliability than the dual-string system.

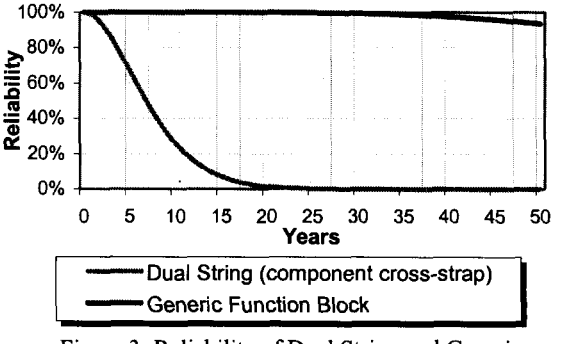

Figure 3: Reliability of Dual String and Generic Function Block Systems

Providing connectivity between the generic blocks and the functional blocks they replace is a difficult implementation problem as it is impossible to determine the required connectivity before launch. Since a *Generic Function Block* has the capability to replace several of the components, regardless of their relative physical locations, the connectivity among the components needs to be flexible. This

may result in different configurations after fault recovery. Though a crossbar type conventional switched network connection might be used, the complexity of such a network can grow very rapidly with the number of functional blocks in the system.

The approach taken in this research is to use wireless interconnection to replace the switched network (see Figure 2). Using a broadcasting wireless network, the connectivity can be simply achieved as long as the distance between two blocks is within the broadcast range. This usually is true for avionics systems as all the components are confined in small space in most cases.

FAULT DIAGNOSIS IN THE ULTRA LONG LIFE AVIONICS SYSTEMS

Since the components in the *Generic Function Block* architecture are not duplicated, it is more difficult to use duplicate-and-compare or voting to detect failures. For some failure modes such as data corruptions, error detection and correction codes are still applicable. On the other hand, function block level failure modes such as crash cannot be detected as directly as the duplicate-and-compare method. The conventional method to solve the problem is to use watchdog timers. However, the detection latency of a watchdog timer is unacceptable in many applications.

The approach this research has taken is to use the autonomous testing methods [2-4]. Autonomous testing is a distributed fault detection technique, in which each component in the system is tested by several other components and the identity of failed components are derived from the test results. As an example, a *Generic Function Block* architecture with three function blocks, as shown in Figure 4, might be considered. Periodically, each function block in Figure 4 tests itself and is tested by another block as shown by arrows (the arrows imply the block C tests the block A, the block A tests the block B and so on). The tests are assumed to be comprehensive enough to detect fault in the tested block. If the testing block is fault-free then it is capable of detecting the fault in the tested block, if any. If, however, the testing block is itself faulty then its testing is unreliable and the test result might not reflect the true status of the tested block. For example, if the block B is faulty, it might find and hence report an incorrect test result that the block C is faulty even when the block C is actually fault-free.

Each block maintains an *opinion register* where it records its opinion, fault-free or faulty, about all the blocks. This opinion might be formed either by direct testing or from the opinion of other blocks. First, each node will perform a self-test. In a fault free situation, each block should pass the self-test. This is shown in Figure 4, where a *'G'* in a cell of the opinion register of a block *x* indicates that the block identified by the cell is fault-free (or "good") in the opinion of the block *x*. In a fault-free situation, since each block should pass the self-test, in each block, the opinion register should have a *'G'* in the cell identifying itself (the contents of other cells are explained later).

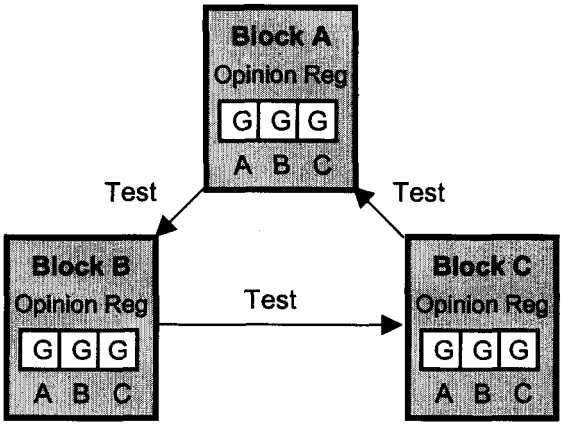

Figure 4: Autonomous Testing System Example

Second, in the test phase, each block is tested by its testers (for the example in Figure 4, each block is tested by another block). For this example, the block A tests the block B, the block B tests the block C, and the block C tests the block A. If there is no fault, each block should also pass each test. This is indicated in Figure 4 where each cell has a *'G'* in the opinion register identified by the block it has tested.

Third, in the diagnosis phase, each block *x* forms its opinion about the health status of any other functional block *y* either from the result of testing, if *x* tested *y*, or from the opinion of some other block *z* such that *z* is healthy in *x*'s opinion. Hence, for the system in Figure 4, even though the

block A has not tested the block C, but by the opinion of the block B (whom the block A finds healthy), the block A finds that the block C is also healthy and uses this information to update its opinion register.

A general model of diagnosis by autonomous testing is represented by a directed graph where each node represents a functional block and each directed edge represents the testing of one block by another. An example is shown in the Figure 5 where each b_i, $i = 0,1,...,6$ represents a functional block and an edge (b_i, b_j) from b_i to b_j represents the testing of the block b_j by the block b_i. The result r_{ij} of testing of the block b_j by the block b_i identifies if b_j passes the test from b_i. If it does then, in b_i's opinion, b_j is fault-free otherwise it is faulty. As mentioned earlier, the considered diagnosis model assumes that the opinion of a fault-free block is correct whereas the opinion of a faulty block is unreliable. It is simple to see that to diagnose the faulty blocks correctly, it is necessary that each block must be tested by at least t other blocks, where t is the largest number of blocks that can fail simultaneously. It has been shown [2] that if the testing between the blocks is arranged as shown in Figure 5, then the faulty blocks can be correctly identified so long as the number of faulty blocks do not exceed two, even though the opinions of the faulty blocks are unreliable.

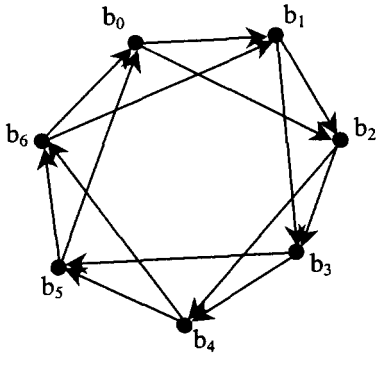

Figure 5: Autonomous Testing Model

Identifying the faulty blocks from the test results can be carried out in a distributed manner as follows. As mentioned earlier, each functional block maintains an opinion register. When a block b_i tests a block b_j and finds it fault-free, b_i also gets the content of the opinion register of the tested block b_j and updates its opinion using the obtained opinion from b_j. On the other hand, if the tested block is found faulty then its opinion register is ignored. It has been shown that such an algorithm, when executed in a distributed manner by all the blocks, results in a correct and consistent identification of faulty blocks by all the fault-free blocks when the testing scheme is as shown in Figure 5. A formal definition of the testing scheme depicted by the Figure 5 is as follows:

- If n is the total number of functional blocks and t is the largest number of blocks that can fail simultaneously then we must have $n \geq 2t+1$
- Each block b_j is tested by a block b_i if and only if $j = i + x \ (mod\ n)$ for $1 \leq x \leq t$.

From this definition, it follows that with additional testing capabilities added to the system in Figure 5, through autonomous testing, up to three faulty functional blocks can be diagnosed.

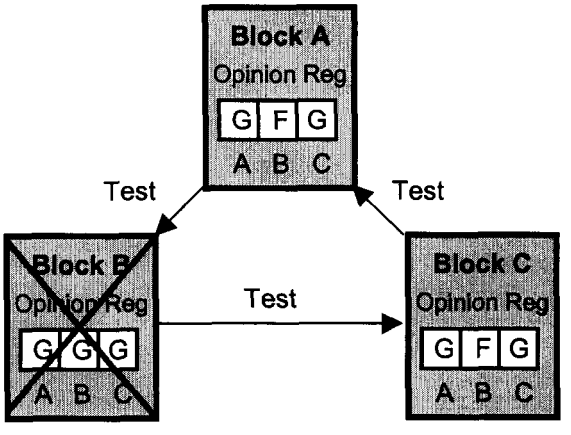

Figure 6: Autonomous Testing System Example with a faulty functional block

The example of Figure 4 is a special case of the above testing capabilities with $n = 3$ and $t = 1$. Consider the case when the functional block B has failed as illustrated in Figure 5. In the test phase, the block A detects the block B's failure and the block C finds out that the block A is healthy. In the diagnosis phase, the block C gets the content of opinion register of the block A and updates its opinion register to infer that the block B has failed. On the other hand, the block A finds the block B faulty and ignores the content of its opinion register. Since it is assumed that there is only one faulty functional block and the block A already knows that the block B has failed, therefore the block A can deduce that the block C is healthy. Both the blocks A and C reach a consistent diagnosis that the block B has failed, so that the

block B will be ignored in subsequent operations. A more general study of the autonomous testing can be found in [2-4].

The inclusion of autonomous test capability needs additional hardware and software features to be included in each functional block and increases the cost of each functional block. For every pair of blocks (b_i, b_j), let h_{ij} denote the cost to implement in b_i the capability of testing b_j. If b_i cannot test b_j then $h_{ij} = \infty$. Let L_j be the set of blocks that can test b_j. Given n functional blocks, in order to diagnose correctly t faulty blocks, at least t of the h_{ij} s must be finite and $|L_j| \geq t$ for all j. Assuming the testing features as given by the Figure 5, we need to find a permutation f on the set {0,1,2,..., n-1}, such that a functional block b_i occupies the position $f(i)$ and the total cost of implementing the testing capability given by

$$\sum_{i=0}^{n-1} \sum_{i \varepsilon L_j} h_{ij}$$
$$f(j)=f(i)+x \ (mod\ n)$$
$$1 \leq x \leq t$$

is minimum.

ULTRA LONG LIFE AVIONICS ARCHITECTURE PROTOTYPE

A prototype of the ultra long life avionics architecture is being developed at the Jet Propulsion Laboratory. This prototype is intended to demonstrate the three key aspects of the architecture: (1) the feasibility of fault recovery with generic function blocks, (2) the feasibility of system reconfiguration with wireless interconnections, and (3) fault diagnosis with distributed autonomous detection.

The "system" that this prototype implements is a navigator that has a gyroscope, star tracker, and accelerometer. Each of the sensors has its own controller, which collect data from the sensor and send it to a local controller for processing. The conceptual design of the prototype is shown in Figure 7.

A Generic Function Block implements each sensor's controller in the prototype. Also an 8051 micro-controller is implemented within the FPGA.

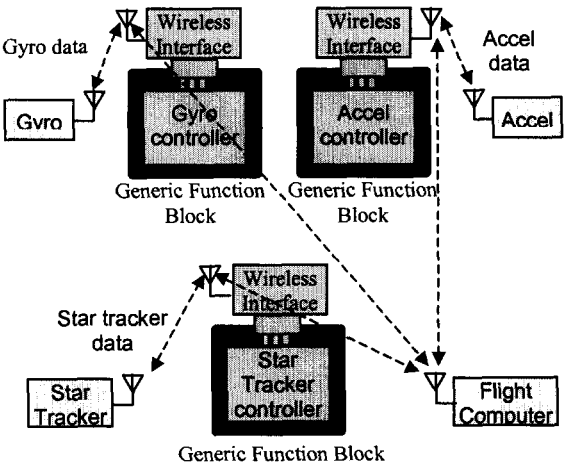

Figure 7: Navigator implemented by Generic Function Block

The *Generic Function Block* contains an FPGA with 600 thousand gates, a wireless interconnection interface, a 64 Kbytes of program memory, and other supporting circuits and displays.

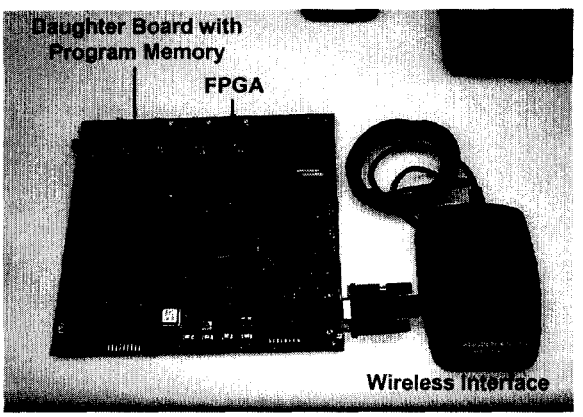

Figure 8: Generic Function Block Implementation

The wireless interconnection is a commercial proprietary protocol, similar to IEEE 802.11b standard [5]. An implementation of the *Generic Function Block* is shown in the Figure 8.

There are three types of software running on the *Generic Function Block*: navigation sensor interface function, autonomous testing, and fault recovery and reconfiguration. Each *Generic Function Block* has the software for all sensor interface controllers but is assigned to run only one type of sensor control software at the system initialization. The autonomous testing software

executes the testing and the diagnosis. The testing performed is a simple reading of a watchdog timer from the *Generic Function Block* under test. The *Generic Function Block* can deduce which block has failed by examining the testing results as described in last section. When a block fails, its function will be assigned to one of the testers of the block. Since every block is tested by at least t other functional blocks, at least one of the testers is guaranteed to be fault-free. In case more than one testers is fault-free, the software for fault recovery and reconfiguration uses a static protocol to determine the functional block to take over.

Due to resource limitations few simplifications are needed to enable the implementation of the prototype in a timely manner. First, since there are no available sensors equipped with the wireless interface, a host computer is used to simulate these sensors. The host computer sends the sensor data to the corresponding interface controller by broadcasting the data along with the sensor name (in form of an address). Only the *Generic Function Block* that has the correct sensor controller will accept and respond to command and data.

Another simplification in this implementation was to use the host computer as a pass-through channel and arbitrator for the communications among the *Generic Function Blocks*. In other words, when a *Generic Function Block* sends a message to another block, it first sends the message to the host computer, which then re-broadcasts the message so that the function block with the correct destination address will receive and respond to the message. The host computer also sets up the *Generic Function Blocks* such that no more than one block will send message at any time. This simplification alleviates the prototype from worrying about the details of the arbitration protocol, so that more effort can be focused on the design of the fault recovery and system reconfigurations. However, this simplification will be removed from future prototypes.

The full testbed with all *Generic Function Blocks* and the host computer is shown in Figure 8. The system integration and test are still underway. When the prototype is competed, the fault recovery with *Generic Function Blocks* can be demonstrated by injecting a fault into one of the blocks (e.g., turning off the power). Then, it is expected that the failure will be detected by the autonomous testing and the task on the failed block will be assumed by its upstream neighbor.

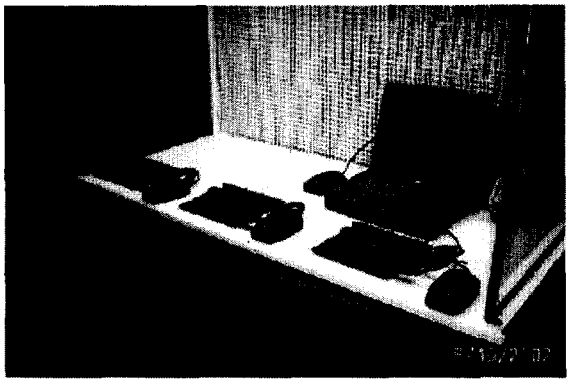

Figure 9: Ultra Long Life Avionics Architecture Testbed

EXPERIMENT OF HARDWARE RECONFIGURATION WITHIN A GENERIC FUNCTION BLOCK

In the prototype, the system reconfiguration and reallocation of functions from one block to another is basically achieved by software. This is possible in the prototype because all the sensor interface controllers have very similar designs. However, in a more general case, blank *Generic Function Blocks* have to be programmed to replace a failed function block. In that case, hardware reconfiguration will be necessary.

The Ultra Long Life Avionics research team at the Jet Propulsion Laboratory has also conducted an experiment to reconfigure *Generic Function Block* through wireless inter-connection. A circuit board was constructed for this experiment as shown in Figure 10. This circuit board also contains an FPGA and a number of I/O interfaces. One of the I/O interface, a parallel port, is modified so that it can accept the configuration for the FPGA from a computer through wireless interconnection. The testbed for the hardware reconfiguration experiment is shown in Figure 11.

The FPGA contains three simple interface circuits: an LED interface, an LCD interface, and a switch interface. In addition, it also dedicates an area of the FPGA as "spare logic" that can be

reprogrammed to replace either the LED or the Switch Interface. The circuit board also includes two sets of switches for fault injection into the LED and Switch Interface circuits. These two interface circuits detect the injected faults by monitoring the positions of these switches. The design of the circuit board and FPGA is depicted in Figure 11.

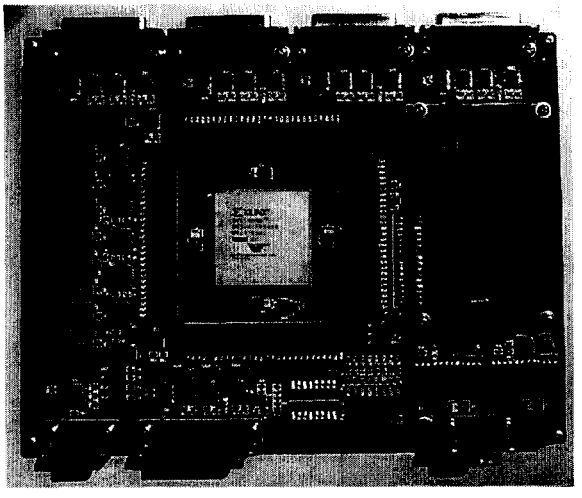

Figure 10: Hardware Reconfiguration Demonstration Circuit Board Design

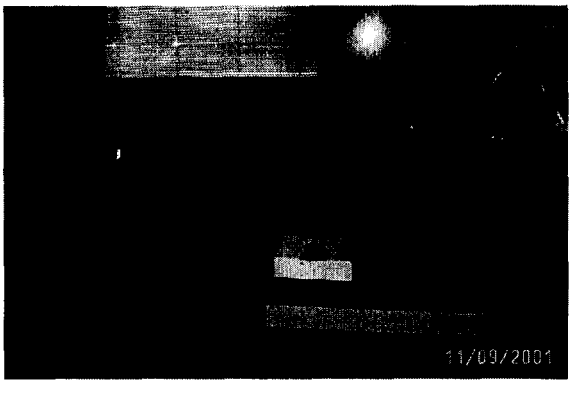

Figure 11: Hardware Reconfiguration Demonstration Testbed

In this experiment, the configuration of the FPGA was first downloaded from the computer to the chip through the wireless interconnection. In normal operation, the System Switches (Figure 12) could be set such that the LED interface could turn on an array of LEDs in different patterns. When a fault was injected into the LED interface, the LEDs would not be turned on properly and an error status signal was sent back to the computer through the wireless interface. Upon receiving the error status, the computer downloaded a new configuration file to the FPGA, again through the wireless interface, so that the spare logic was used to replace the LED interface. Similarly, when a fault was injected to the Switch Interface (Figure 11), the System Switches would not function properly and an error status was sent to the computer. Consequently, a new configuration file was downloaded to the FPGA, so that the spare logic was used to replace the Switch Interface. This experiment was demonstrated successfully in the testbed.

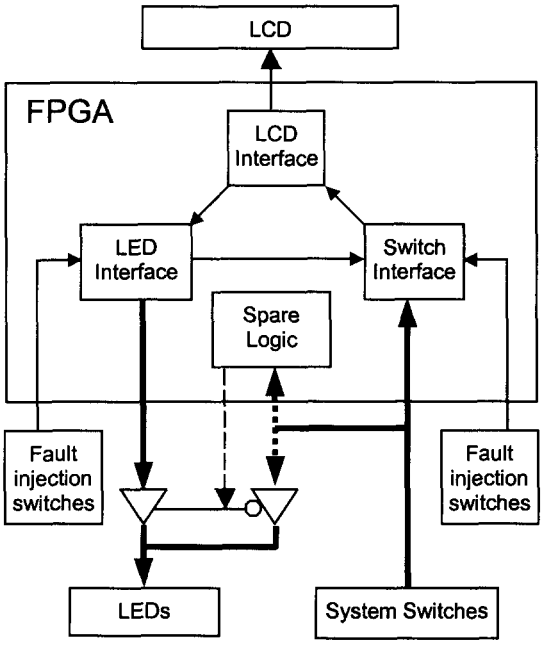

Figure 12: FPGA Design for Hardware Reconfiguration Demonstration

This hardware reconfiguration capability will be incorporated in future Ultra Long Life Avionics experiments.

CONCLUSION

This paper has explored some unconventional architecture design techniques that utilize much less redundant components for sustaining very long duration missions. An architecture based on the concept of *Generic Function Blocks* has been developed. A prototype of this architecture representing a navigator subsystem is being constructed at the Jet Propulsion Laboratory. This prototype can detect failures in any one of the *Generic Function Blocks* by means of

Autonomous Testing through wireless communication among the blocks. Once the failed block is identified, software technique is employed to relocate the tasks of the failed block to a healthy block.

An independent experiment of reconfiguring the hardware design of an FPGA through wireless interconnection has also been conducted. The experiment was conducted successfully and the technique will be incorporated into future system reconfiguration/recovery experiments.

FUTURE WORK

Many issues of this ultra long life avionics architecture have not bee addressed by the experiments mentioned above. Examples of these issues are:

1. The wireless interface in this prototype has to handle only a few function blocks, and the arbitration problem is simplified by the host computer. In the next step, a more sophisticated wireless protocol should be developed so that large number of *Generic Function Blocks* should be able to communicate directly with each other simultaneously.
2. A more realistic testbed need to be constructed, in which all sensors have wireless interface and can communicate with any *Generic Function Blocks* directly.
3. A "self-repair" capability should be developed in each Generic Function Block, so that a failed block can be salvaged and reused.
4. The wireless interface among the Generic Function Blocks might interfere with the telecommunication system or other on-board electronics. The effect of the wireless interface need to be investigated and design techniques should be developed to minimize such interference.
5. Traditional verification techniques such as accelerated life test might be too expensive or taking too long to verify the reliability and lifetime of the Ultra Long Life Avionics architecture. New verification techniques have to be developed for systems that are ultra long life.

ACKNOWLEDGEMENT

The research described in this paper was carried out by the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration.

REFERENCES

[1] S. Chau and J. Blosiu, "Ultra Long Life System Concept, Rev 1," Internal Document, Jet Propulsion Laboratory, Feb 16, 2001.

[2] A. Sengupta and C. Rhee, "Different classes of diagnosable systems: relationship and common diagnosis algorithm", IEEE Trans. Circuits and Systems, vol. 38, pp. 642-645, June, 1991.

[3] A. Sengupta and A. Sen, "On the diagnosability problem for a general model of diagnosable systems", *Information Science*, vol. 42, pp. 83-94, 1987.

[4] A. Sengupta and C. Rhee, "On a generalization of self-implicating structures in diagnosable systems", *IEEE Trans. Circuits and Systems*, vol. 40, no. 4, pp. 239-245, April, 1993.

[5] IEEE std 802.11b-1999, "Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) specifications: Higher-Speed Physical Layer Extension in the 2.4 GHz Band